Universitat de Lleida
Escola Politècnica Superior
Enginyeria Tècnica en Informàtica de Sistemes

Trabajo de final de carrera

Integración de Google Maps y TSP Solvers para la definición y resolución del TSP

Autor: Miquel Vidal Morales
Director: Carlos José Ansótegui Gil

25 de septiembre de 2012

Índice general

Índice de figuras

Capítulo 1

Introducción

Este trabajo final de carrera presenta la arquitectura e implementación de un entorno web para la descripción y visualización de instancias reales del *TSP* (*Travelling Salesman Problem*), a través de *Google Maps*, y su posterior resolución mediante técnicas de optimización combinatoria.

El problema del viajante de comercio (TSP), consiste en que dadas n ciudades, el objetivo es encontrar una ruta que, comenzando y terminando en la misma ciudad, sea capaz de pasar exactamente una vez por cada una de las demás ciudades, asegurando que la distancia recorrida por el viajante sea la mínima. El problema reside en el número de posibles combinaciones, el cual viene dado por el factorial de ciudades ($n!$), lo que significa que la solución por fuerza bruta sea impracticable incluso para valores de n moderados, con los medios computacionales de los que actualmente disponemos. Esto demuestra el por qué el TSP es un problema *NP-Completo*.

Para que la descripción de instancias TSP y la resolución de las mismas resultaran más intuitivas, se ha diseñado un entorno web, en donde se utilizan herramientas y funcionalidades proporcionadas por Google Maps.

Una vez formulada una instancia TSP, se enviarán los datos necesarios para resolver el problema, mediante el protocolo *XML-RPC*, a un servidor que será el responsable de codificar el problema en MaxSAT e invocar a un resolutor con el objetivo de que éste resuelva el problema.

XML-RPC es un protocolo de llamada a procedimiento remoto (*Remote Procedure Call*) que utiliza XML para codificar los datos y HTTP como protocolo de transmisión. Existen distintas implementaciones para varios lenguajes de programación. En este proyecto se ha utilizado una implementación sobre PHP, la librería *XML-RPC for PHP*, disponible en `http://phpxmlrpc.sourceforge.net`.

El protocolo XML-RPC nos permite separar los servicios de definición y visualización del problema, del proceso de resolución del mismo, por lo que si alguien quisiera resolver una instancia del TSP, solo tendría que llamar a la función correspondiente del servidor con los parámetros adecuados para obtener la solución al problema, sin la necesidad de desarrollar una herramienta capaz de definir y visualizar instancias del TSP. Por el contrario, si alguien deseara resolver el problema mediante otras técnicas distintas a las que se utilizan en este proyecto, tendría a su disposición una herramienta que le permitiría definir y visualizar instancias del TSP.

El problema *MaxSAT* es una generalización del problema de satisfactibilidad booleana (*SAT*) y consiste en dado un conjunto de cláusulas booleanas, determinar el número máximo de cláusulas que se pueden satisfacer, ya que no siempre en todos los casos se podrán cumplir todas las restricciones de un problema. El problema MaxSAT es un problema natural combinatorio que puede ser utilizado en distintos ámbitos, tales como: combinatorial auctions, planificación, creación de horarios, FPGA routing, instalación de paquetes de software, etc...

El solver que se ha utilizado para resolver las instancias de MaxSat es el *WMaxSatz.*

1.1. Objetivos

El objetivo principal de este proyecto es proporcionar un entorno web para facilitar la descripción y visualización de instancias TSP, así como su posterior resolución.

Concretamente, se propone:

- Crear un entorno web que ofrezca los siguientes servicios:
  - Definir instancias del TSP y visualizar su solución a través de Google Maps
  - Resolver instancias del TSP
- Proveer una arquitectura que permita intercambiar y extender los servicios de visualización y resolución estableciendo su comunicación mediante XML-RPC
- Resolver instancias TSP mediante su reformulación al problema MaxSAT.
- Evaluar el rendimiento de nuestra aplicación en comparación con otros servicios web similares

1.2. Estructura del documento

El presente trabajo consta de 6 capítulos.

En el primer capítulo se presentan los objetivos principales del proyecto, así como la estructura que sigue este documento.

El capítulo 2 empieza definiendo el TSP. Seguidamente, se explica en que consiste la comunicación mediante el protocolo XML-RPC y cuales son los servicios, proporcionados por Google Maps, utilizados en este proyecto. Finalmente se describen los problemas SAT, MaxSAT, y Weighted Partial MaxSAT, además de los distintos tipos de resolutores que existen para resolver instancias MaxSAT.

En el capítulo 3 se describe la reformulación que se ha llevado a cabo del TSP, hacia el problemas MaxSAT, con el objetivo de que posteriormente los resolutores de MaxSAT sean capaces de interpretar y resolver el problema.

El capítulo 4 se encarga de describir las funcionalidades del entorno web creado para definir, visualizar y resolver instancias del TSP, así como de explicar la comunicación mediante XML-RPC que se ha utilizado para enlazar la parte que permite definir y visualizar instancias del TSP en mapas reales, de la parte que se encarga de resolver el problema.

En el capítulo 5 se presentan los resultados experimentales obtenidos de resolver el TSP mediante el resolutor WMaxSatz. También se presenta una comparativa en la que se evalúa el rendimiento del entorno web presentado en este proyecto, comparandolo con otras aplicaciones similares, disponibles en Internet.

Finalmente, en el capítulo 6, se exponen las conclusiones del presente trabajo y se plantea cuales son las líneas de futuro trabajo.

Capítulo 2

Estado del arte

En este capítulo se definen el TSP, así como los problemas SAT y MaxSAT a la vez que se explica el funcionamiento del protocolo XML-RPC y de las distintas herramientas que nos proporciona Google Maps para poder trabajar con mapas reales.

En primer lugar, se pasa a definir el TSP.

Seguidamente, se introducen dos herramientas fundamentales para el correcto desarrollo de este proyecto, el API de Google Maps y el protocolo XML-RPC. Veremos algunos de los servicios que nos proporciona Google para trabajar con mapas reales, a la vez que explicaremos cuales son las características y funcionalidades del protocolo XML-RPC.

Finalmente se habla de las codificaciones que se han utilizado para resolver el TSP. En primer lugar, se define formalmente el problema SAT. Seguidamente, se pasa a definir el problema MaxSAT, que es la variante de optimización del problema SAT. En concreto, en este proyecto, haremos hincapié en el problema Weighted Partial MaxSAT, el cual es aún más general que MaxSAT.

2.1. TSP (Travelling Salesman Problem)

El TSP se puede definir de la siguiente forma. Sean n ciudades de un territorio, el objetivo es encontrar una ruta que, comenzando y terminando en una ciudad concreta, pase exactamente una vez por cada una de las ciudades y minimice la distancia recorrida por el viajante.

En realidad se trata de encontrar una permutación

$$P = \{c_0, c_1, ..., c_{n-1}\}$$

tal que

$$d\rho = \sum_{i=0}^{N-1} d[c_i, c_{i+1 mod(N)}]$$

sea mínima, donde $d[x, y]$ representa la distancia que hay entra la ciudad X y la ciudad Y. En un TSP simétrico, la distancia entre dos ciudades es la misma en cada dirección, formando un grafo no dirigido. Esta simetría reduce a la mitad el número de posibles soluciones. En el TSP asimétrico, no necesariamente hay caminos en ambas direcciones y las distancias pueden ser diferentes, formando un grafo dirigido. En este documento, cuando nos referimos a TSP, siempre estaremos hablando de TSP asimétrico.

Una formulación equivalente en términos de la teoría de grafos es la de encontrar, en un grafo completamente conexo y con arcos ponderados, el ciclo hamiltoniano de menor coste. En esta formulación cada vértice del grafo representa una ciudad, cada arco representa un camino y el peso asociado a cada arco representa la longitud del camino.

2.2. Google Maps API

El API de Google Maps, implementado en JavaScript, proporciona diversas utilidades para manipular mapas y para añadir contenido al mapa mediante diversos servicios. Concretamente, en este proyecto se ha utilizado la versión 3 del API, la cual está especialmente diseñada para proporcionar una mayor velocidad. La versión 3 de Google Maps JavaScript API es un servicio gratuito disponible para cualquier sitio web que sea gratuito para el consumidor. No obstante, veremos a lo largo de esta sección, que existe alguna que otra restricción a la hora de utilizar algunos de los servicios que nos proporciona.

A continuación, introduciremos algunos de los objetos y servicios proporcionados por Google Maps, utilizados en este proyecto:

- ***Map*** (*google.maps.Map*): el elemento fundamental en cualquier aplicación del API de Google Maps v3. La figura 2.2.1 nos muestra como inicializar un mapa, para poder empezar a trabajar con él. Para ello, deben seguirse las siguientes instrucciones:

```
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="initial-scale=1.0, user-scalable=no" />
<style type="text/css">
  html { height: 100% }
  body { height: 100%; margin: 0px; padding: 0px }
  #map_canvas { height: 100% }
</style>
<script type="text/javascript"
    src="https://maps.google.com/maps/api/js?sensor=set_to_true_or_false">
</script>
<script type="text/javascript">
  function initialize() {
    var latlng = new google.maps.LatLng(-34.397, 150.644);
    var myOptions = {
      zoom: 8,
      center: latlng,
      mapTypeId: google.maps.MapTypeId.ROADMAP
    };
    var map = new google.maps.Map(document.getElementById("map_canvas"),
        myOptions);
  }

</script>
</head>
<body onload="initialize()">
  <div id="map_canvas" style="width:100%; height:100%"></div>
</body>
</html>
```

Figura 2.2.1: Declaración de un mapa mediante el API de Google Maps

- Declarar la aplicación como HTML5 mediante la declaración <!DOCTYPE html>.
- Incluir el código JavaScrit del API mediante la etiqueta *script.*
- Alojar el mapa en un elemento * .*
- Crear un objeto JavaScript para alojar una serie de propiedades del mapa.
- Crear una función JavaScript para crear un objeto de mapa.
- Inicializar el mapa desde el evento *onload* de la etiqueta *<body>* de nuestra página.

- ***Latitudes y longitudes***: necesitamos una manera de hacer referencia a ubicaciones del mapa. El objeto *google.maps.LatLng* proporciona esta posibilidad dentro del API de Google Maps. Los objetos LatLng se construyen transmitiendo sus parámetros en el orden habitual {latitud, longitud}. Los objetos LatLng tienen muchas aplicaciones en el API de Google Maps. Por ejemplo, el objeto *google.maps.Marker* utiliza un elemento LatLng en su constructor y coloca una superposición de marcador en la ubicación geográfica especificada del mapa.

- ***Ubicación geográfica***: hace referencia a la identificación de la ubicación geográfica de un usuario o dispositivo informático a través de diversos mecanismos de recopilación de datos. Normalmente, la mayoría de los servicios de ubicación utilizan direcciones de enrutamiento de red o dispositivos GPS internos para determiar esta ubicación.

- ***Localización de idioma***: el API de Google Maps utiliza la configuración de idioma preferida del navegador al mostrar información de texto como, por ejemplo, los nombres de los controles, las notificaciones de derechos de autor, las indicaciones para llegar en coche y las etiquetas de los mapas.

- ***Eventos***: JavaScript en el navegador está *orientado a eventos,* lo que significa que JavaScript responde a las interacciones generando eventos y espera que un programa detecte los eventos interesantes. Cada objeto del API exporta una determinada cantidad de eventos con nombres. Los programas interesados en determinados eventos registran *detectores de eventos* de JavaScript para estos eventos y ejecutan código al recibirlos mediante el registro de controladores de eventos *addListener()* en el espacio de nombres *google.maps.event.* Hay dos tipos de eventos:

  - Los eventos de usuario se propagan desde el DOM hasta el API de Google Maps. Estos eventos son distintos e independientes de los eventos DOM estándar.
  - Las notificaciones de cambio de estado de MVC reflejan los cambions en los objetos del API de Google Maps y se denominan mediante una convención *property_ changed.*

- ***Controles***: los mapas de Google Maps contienen elementos de interfaz de usuario que permiten al usuario interactuar con el mapa. Estos elementos se denominan controles. El API de Google Maps dispone de varios controles integrados que puedes emplear en tus mapas:

    - El control de zoom muestra un control deslizante o botones "+/-" para controlar el nivel de zoom del mapa.
    - El control de desplazamiento muestra botones para desplazarse por el mapa.
    - El control MapType permite al usuario alternar entre los diferentes tipos de mapa, como mapa de carreteras y satélite.

- ***Superposiciones***: son objetos del mapa que están vinculados a coordenadas de latitud y longitud, por lo que se mueven al arrastrar o aplicar zoom sobre el mapa. Las superposiciones son los objetos que se añaden al mapa para designar puntos, líneas, áreas o grupos de objetos.

- ***Servicio de rutas*** (*google.maps.DirectionsService*): este servicio nos permite calcular y visualizar rutas para llegar de una ubicación a otra, usando distintos métodos de transporte. La ruta se muestra como una polilínea en el mapa. Las rutas pueden especificar los orígenes, los destinos y los hitos como cadenas de texto o como coordenadas de latitud/longitud. El uso de este servicio está sujeto a las siguientes restricciones:

    - 10 puntos por ruta. Podemos evitar esta restricción creando múltiples rutas para que posteriormente en el mapa aparezcan como una sola.
    - 2.500 solicitudes de indicaciones al día

- ***Servicio de distancias*** (*google.maps.DistanceMatrixService*): el API de matriz de distancia de Google es un servicio que proporciona el tiempo y la distancia de viaje para una matriz de orígenes y destinos. La información devuelta se basa en la ruta óptima entre los puntos de partida y llegada, según los cálculos del API de Google Maps, y se compone de dos filas que incluyen los valores de duración y de distancia de cada par de puntos. No obstante, para un uso gratuito, existen algunas limitaciones:

    - 100 elementos (distancia entre 2 puntos) por cada 10 segundos. Esto puede ocasionar un tiempo de espera extra en nuestro caso.
    - 100 elementos como máximo por cada petición, lo que significa que debemos controlar el producto de los orígenes por las destinaciones para que la matriz resultante no contenga más de 100 elementos.
    - 2.500 elementos al día

2.3. El protocolo XML-RPC

XML-RPC (*Extensible Markup Language – Remote Procedure Call*) es un protocolo de llamada a procedimiento remoto que usa XML para codificar los datos y HTTP como protocolo de transporte. En concreto, utiliza peticiones POST de HTTP para enviar un mensaje, en formato XML, señalando:

- El procedimiento que se va a ejecutar en el servidor
- Los parámetros utilizados por dicho procedimiento

Este protocolo fue creado por Dave Winer en 1998. La simplicidad del XML-RPC contrasta con otros protocolos que tienen una curva de aprendizaje mucho mas pronunciada, tales como SOAP. Una de las características más importantes de este protocolo es que permite que las aplicaciones que lo utilizan, puedan comunicarse independientemente de cual sea el sistema operativo o el lenguaje de programación que utilicen. Es por esto, que un mismo servidor, puede recibir peticiones de distintas aplicaciones implementadas en múltiples lenguajes de programación.

Los tipos de datos que podemos utilizar en los mensajes XML-RPC son los siguientes:

- Array: <array>
- Base64: <base64>
- Boolean: <boolean>
- Date/Time <dateTime.iso8601>
- Double: <double>
- Integer: <i4> o <int>
- String: <string>
- Struct <struct>
- Null: <nil/>

Una petición de XML-RPC está formada por un mensaje XML con la siguiente estructura:

```
<?xml version =”1.0”?>
<methodCall>
```

```
    <methodName></methodName>
    <params>
        <param>
            <value></value>
        </param>
    </params>
</methodCall>
```

El mensaje contiene un único elemento $<methodCall>$ que a su vez, incorpora los elementos $<methodName>$, el cual contiene el nombre del método que debe ejecutarse en el servidor, y $<params>$, formado por tantos elementos $<param>$ como parámetros tenga el método que estamos invocando.

La respuesta, por su parte, está formada por un mensaje XML que contiene un único elemento, $<methodResponse>$, el cual puede contener distintos elementos, dependiendo de si el mensaje devuelve una respuesta correcta o un error.

En caso de que se trate de una respuesta válida, el elemento $<methodResponse>$ contendrá un único elemento $<params>$ el cual contendrá un único elemento $<param>$, que finalmente contendrá un único elemento $<value>$, en el que podremos encontrar la respuesta a nuestra petición.

```
<?xml version =”1.0”?>
<methodResponse>
    <params>
        <param>
            <value></value>
        </param>
    </params>
</methodResponse>
```

Si por el contrario, el mensaje de respuesta contiene errores, el elemento $<methodResponse>$ estará formado por un único elemento $<fault>$, que a su vez contendrá un único elemento $<value>$, el cual es un struct que dispondrá dos elementos, un elemento $<int>$ y un elemento $<string>$, que nos indican el identificador y la descripción del error, respectivamente.

```
<?xml version =”1.0”?>
<methodResponse>
```

```
    <fault>
        <value>
            <struct>
              <member>
                <name></name>
                <value><int></int></value>
              </member>
              <member>
                <name></name>
                <value><string></string></value>
              </member>
            </struct>
        </value>
    </fault>
</methodResponse>
```

2.4. Los problemas SAT y MaxSAT

En primer lugar, es necesario definir formalmente una serie de conceptos básicos relativos al problema SAT. Dado un conjunto infinito numerable de variables proposicionales $\mathcal{X}$:

- Un *literal* l es una variable $x_i \epsilon \mathcal{X}$ o su negación $\neg x_i$.
- Una *cláusula* C es un conjunto de literales, también denotada como $l_1 \vee ... \vee l_r$ o $\square$ si se trata de una *cláusula vacía*.
- Una *formula SAT* φ es un conjunto de cláusulas, también denotada como $C_1 \wedge ... \wedge C_m$. Las fórmulas que consisten en una conjunción finita de cláusulas, cada una de ellas consistente en una disyunción finita de literales, se dice que están en formato *CNF* (*Conjunctive Normal Form*) . El conjunto de variables que ocurren en una fórmula se denota como var(φ).

Una *asignación de verdad* es una función $I : X \to \{0, 1\}$, donde, $X \subset \mathcal{X}$. Esta función puede ser extendida a los literales, cláusulas y fórmulas SAT:

- Para los literales, $I(\neg x_i) = 1 - I(x_i)$, si $x_i \epsilon X$; en caso contrario, $I(l) = l$.
- Para las cláusulas, $I(l_1 \vee ... \vee l_r) = I(l_1) \vee ... \vee I(l_r)$, considerando las simplificaciones $1 \vee C = 1, 0 \vee C = C$ y $\square = 0$.

- Para las fórmulas, $I(C_1 \wedge ... \wedge C_r) = I(C_1) \wedge ... \wedge I(C_r)$, considerando las simplificaciones $1 \wedge \varphi = \varphi, 0 \wedge \varphi = 0$ y $\varnothing = 1$.

Una asignación de verdad I *satisface* un literal, cláusula o fórmula si le asigna 1, y lo *falsifica* si le asigna 0. Una asignación I que satisface una fórmula es un *modelo* para esa formula.

Una fórmula es *satisfactible* si existe una asignación de verdad que la satisface. En caso contrario, es *insatisfactible*.

El *problema SAT* consiste en determinar si una fórmula SAT es satisfactible o no.

Dada una formula SAT φ insatisfactible, un *núcleo de insatisfactibilidad* φ_c es un subconjunto de cláusulas $\varphi_c \subseteq \varphi$ que es también insatisfactible. Un *núcleo de insatisfactibilidad minimal* es un núcleo de insatisfactibilidad tal que cualquier subconjunto propio es satisfactible.

Para resolver el problema SAT son necesarios los resolutores SAT. Estos resolutores pueden ser completos o incompletos:

- **Resolutores completos**: hacen una búsqueda ordenada por todo el espacio de interpretaciones. Cuando terminan la búsqueda, o bien se ha encontrado una solución, o bien se ha explorado todo el espacio sin encontrar ninguna solución.
- **Resolutores incompletos**: no exploran todo el espacio de interpretaciones. Su ejecución termina cuando se encuentra una solución o bien si se ha superado una cota de tiempo establecida sin haber encontrado ninguna solución.

El *problema MaxSAT* es una generalización del problema de satisfactibilidad booleana (SAT) y consiste en determinar el número máximo de cláusulas que se pueden satisfacer en una fórmula SAT. Esto significa que no siempre en todos los problemas, se podrán satisfacer todas las cláusulas. Dicho esto, podemos dividir las cláusulas en 2 grupos:

- ***hard***: cláusulas o restricciones que deben ser satisfechas
- ***soft***: cláusulas o restricciones que pueden o no ser satisfechas
  En este grupo, podemos asignar distintos pesos a las cláusulas (*weights*), donde el peso es el grado de penalización asociado a incumplir la cláusula. Esto es útil para indicar la importancia de unas u otras cláusulas.

Existen distintos tipos de instancias MaxSAT, pero en este documento se habla solamente del *problema Weighted Partial MaxSAT*, que a su vez, es una generalización del problema MaxSAT. Diremos que una instancia es *weighted* cuando tenga pesos asignados a las cláusulas, y diremos que se trata de una instancia *partial* cuando las cláusulas estén divididas en *hard* y *soft*.

En este punto, podemos decir que dada una instancia *Weighted Partial MaxSat,* queremos encontrar una asignación de valores que cumpla todas las cláusulas *hard,* y que a su vez minimice la suma de los pesos de las cláusulas *soft* incumplidas. Dicha asignación de valores se considera la óptima en este contexto.

No obstante, para definir correctamente el problema Weighted Partial MaxSat es necesario introducir los siguientes conceptos:

- Una *cláusula weighted* es un par (C, ω), donde C es una cláusula y ω puede ser tanto un número natural como infinito, que indica el coste de falsificar C. Si una fórmula contiene cláusulas weighted, se la denomina *fórmula Weighted*. Una cláusula es *hard* si su correspondiente peso es infinito, en cualquier otro caso la cláusula se considera *soft*.
- Una *fórmula Weighted Partial MaxSAT* es un conjunto de cláusulas weighted

  $\varphi = \{(C_1,\omega_1), ..., (C_m, \omega_m), (C_{m+1}, \infty), ..., (C_{m+m'},\infty)\}$

  donde las primeras m cláusulas son *soft* y las últimas m' cláusulas son *hard*. En las cláusulas soft el par (C, ω) es equivalente a tener ω copias de la cláusula $(C, 1)$ en el conjunto. En las cláusulas hard el peso correspondiente es infinito, ya que deben ser siempre satisfechas.
- El *coste* sobre una fórmula Weighted Partial MaxSAT φ de una asignación I es la suma de los pesos de las cláusulas falsificadas por I:

  $$\sum_{\substack{(C_i,\omega_i)\epsilon\varphi \\ I(C_i)=0}} \omega_i$$

- El *coste óptimo* de una fórmula Weighted Partial MaxSAT es el coste mínimo de todas las asignaciones de sus variables:

  $cost(\varphi) = mín(cost(\varphi, I) \parallel I : var(\varphi) \longrightarrow \{0, 1\}$

  Una *asignación óptima* es una asignación con coste óptimo.

El *problema Weighted Partial MaxSAT* consiste en determinar el coste óptimo de una fórmula Weighted Partial MaxSAT. Se puede formular también como un problema de maximización de una función lineal:

$$maximizar \sum_{i=1}^{m} b_i \cdot \omega_i \tag{2.4.1}$$

sujeto a un conjunto de restricciones:

$$\bigwedge_{i=1}^{m} b_i \longleftrightarrow C_i \tag{2.4.2}$$

$$\bigwedge_{j=m+1}^{m'} C_j \tag{2.4.3}$$

donde b_i son variables booleanas, 2.4.1 es la función lineal a maximizar, 2.4.2 asegura que b_i es cierta si, y sólo si, C_i es evaluada a cierto y 2.4.3 es el conjunto original de cláusulas hard.

Para expresar este problema de maximización como uno de equivalente de minimización, solamente es necesario reemplazar 2.4.1 por:

$$minimizar \sum_{i=1}^{m} \omega_i - \sum_{i=1}^{m} b_i \cdot \omega_i$$

MaxSAT Solvers

El problema MaxSAT es un problema combinatorio natural que se puede encontrar en ámbitos tan diversos como subastas, planificación, horarios o enrutamiento. Es un problema NP-duro, por lo que los objetivos se centran en diseñar e implementar solvers eficientes para tratar problemas reales o industriales, que no son de los más duros. A fin de comparar la eficiencia de los distintos MaxSAT solvers, cada año tiene lugar la MaxSAT Evaluation organizada por Josep Argelich de la Universitat de Lleida, Chu Min Li de la Université de Picardie, Felip Manyà del IIIA-CSIC y Jordi Planes de la Universitat de Lleida. Las fórmulas están divididas en tres familias:

- Industrials (industriales)
- Random (aleatorios)
- Crafted (artificiales)

Los solvers también se dividen en varias categorías:

- MS (MaxSAT)
- PMS (Partial MaxSAT)
- WMS (Weighted MaxSAT)
- WPMS (Weighted Partial MaxSAT)

Hay principalmente dos tipos de solvers para las diferentes variantes de optimización de MaxSAT citadas anteriormente:

- Solvers basados en ramificación y poda (Branch and Bound): WMaxSatz [Li et al., 2009], MiniMaxSat [Heras et al., 2007], IncWMaxSatz [Lin et al., 2008].
- Solvers basados en tests de satisfactibilidad (SAT Testing): SAT4J [Berre, 2001], MiniSat$^+$ [Eén and Sörensson, 2006], WPM1 [Ansotegui et al., 2009], WBO y Msuncore [Manquinho et al., 2009, Manquinho et al., 2010] y WPM2 [Ansotegui et al., 2010].

Del análisis de los resultados de la MaxSAT Evaluation [Argelich et al., 2008] se puede extraer que los solvers basados en ramificación y poda son por lo general más eficientes con las fórmulas aleatorias y algunas artificiales, y los basados en tests de satisfactibilidad con las fórmulas industriales.

A continuación, se describe el funcionamiento de los solvers de este segundo grupo para el problema Weighted Partial MaxSAT. Se basa en encontrar el coste óptimo k_{opt} de una fórmula Weighted Partial MaxSAT

$$\varphi = \{(C_1, \omega_1), ..., (C_m, \omega_m), (C_{m+1}, \infty), ..., (C_{m+m'}, \infty)\}$$

comprobando iterativamente la satisfactibilidad de fórmulas SAT. Estas fórmulas SAT serán de la forma φ_k, que es satisfactible si y sólo si, φ tiene una asignación de coste menor o igual a k. Para codificar φ_k puede ampliarse cada cláusula soft C_i con una nueva variable booleana auxiliar b_i, y añadir la conversión a CNF de la *restricción lineal pseudo-booleana* correspondiente a que el sumatorio de los productos de las variables booleanas auxiliares b_i y los pesos de sus cláusulas ω_i es menor o igual a k:

$$\varphi_k = \{C_1 \vee b_1, ..., C_m \vee b_m, C_{m+1}, ..., C_{m+m'}\} \cup CNF(\sum_{i=1}^{m} b_i \cdot \omega_i \leq k)$$

φ_k es satisfactible para $k \geq k_{opt}$ e insatisfactible para $k \leq k_{opt}$. La búsqueda del coste óptimo de φcorresponde a la ubicación exacta de esta transición de fase entre fórmulas satisfactibles e insatisfactibles. Se puede realizar siguiendo diferentes estrategias:

- Desde $k = 0$ aumentando k hasta que φ_k sea satisfactible.
- Desde $k = \sum_{i=1}^{m} \omega_i$ disminuyendo k hasta que φ_k sea insatisfactible.
- Alternando φ_k insatisfactibles y satisfactibles hasta que el algoritmo converge a k_{opt}. Por ejemplo utilizando un esquema de búsqueda binaria en que la nueva k se sitúa exactamente en la media aritmética entre la k insatisfactible más alta y la k satisfactible más baja encontradas hasta el momento.

La clave para impulsar la eficiencia de estas estrategias es aprovechar la información adicional de la ejecución del SAT solver para las fórmulas φ_k en las siguientes iteraciones. Dos de los solvers que empezaron a explorar este camino son SAT4J [Berre, 2001] y MiniSat$^+$[Eén and Sörensson, 2006]. Este segundo solver inicia la búsqueda con $k = \sum_{i=1}^{m} \omega_i$ y cada vez que el SAT solver devuelve satisfactible, usa el refinamiento siguiente. Comprueba la asignación I que ha satisfecho φ_k y asigna el siguiente k igual a la suma de los pesos de las cláusulas soft con variables auxiliares evaluadas a cierto, menos uno $k = \sum_{I(b_i)=1} \omega_i - 1$. Si el SAT solver devuelve insatisfactible, entonces el algoritmo se detiene y el coste óptimo es $k + 1$.

Otro refinamiento posible consiste en el algoritmo de Fu y Malik [Fu, 2007, Fu and Malik, 2006], descrito originalmente para el problema Partial MaxSAT. Se inicia la búsqueda con $k = 0$ y aumenta este valor hasta que φ_k es satisfactible. Mientras φ_k es insatisfactible, el solver SAT devuelve también un núcleo de insatisfactibilidad no necesariamente mínimo. Los siguientes φ_k se construyen añadiendo variables auxiliares a las cláusulas soft que pertenecen al núcleo, y *restricciones de cardinalidad* sobre estas variables indicando que al menos una de ellas tiene que ser cierta para evitar que el solver SAT vuelva a encontrar el mismo núcleo de insatisfactibilidad. Los solvers WPM1 [Ansotegui et al., 2009], WBO y Msuncore [Manquinho et al., 2009, Manquinho et al., 2010] se basan en la extensión de este algoritmo para el problema Weighted Partial MaxSAT. La adición de nuevas restricciones de cardinalidad en cada iteración, permite resolver de manera más eficiente las fórmulas φ_k insatisfactibles. Sin embargo, la acumulación de muchas variables auxiliares en las cláusulas soft puede mermar la eficiencia del solver. Este problema se soluciona en el algoritmo WPM2 [Ansotegui et al., 2010].

Formato del problema

Los problemas MaxSAT deben seguir la siguiente estructura:

- Problema MaxSAT en formato DIMACS.

  - El fichero puede empezar con comentarios, los cuales se identifican empezando la línea con el carácter *'c'*.
  - Seguidamente, encontramos la línea *p cnf nbvar nbclauses* la cual nos indica que se trata de una instancia CNF. Por su parte, *nbvar* nos indica el número de variables de que dispone la instancia, mientras que *nbclauses* se refiere al número de cláusulas.
  - A continuación, cada línea de texto representa una cláusula en forma de secuencia de números distintos de cero comprendidos entre *-nbvar* y *+nbvar* terminando la línea siempre con el carácter 0, indicando este el fin de la cláusula. Los números positivos nos indican a que variable nos estamos refiriendo, mientras que los negativos, nos indican lo mismo, pero negando la variable.

    ```
    c
    c comments MaxSAT
    c
    p cnf 3 4
    1 -2 0
    -1 2 -3 0
    -3 2 0
    1 3 0
    ```

- Problema Weighted MaxSAT

  - La única diferencia con el problema MaxSAT en formato DIMACS es que en este caso las cláusulas tienen asociado un peso, el cual esta representado con el primer entero, mayor o igual a 1 y menor que 2^{63}, que aparece en cada línea que represente una cláusula. Para indicar al resolutor que se trata de un problema Weighted MaxSAT debemos cambiar la línea de parámetros para que sea de la forma *p wcnf nbvar nbclauses,* indicando *wcnf* que estamos en un problema Weighted MaxSAT.

    ```
    c
    c comments Weighted MaxSAT
    c
    p wcnf 3 4
    20 1 -2 0
    5 -1 2 -3 0
    7 -3 2 0
    4 1 3 0
    ```

- Problema Partial MaxSAT

- La línea de parámetros de este problema tiene la forma *p wcnf nbvar nbclauses top*. En este problema se dividen las cláusulas en 2 grupos, hard y soft. Las cláusulas hard tienen el peso indicado en la línea de parámetros por la variable *top,* mientras que las cláusulas soft tienen un peso de 1. Para que el problema este formulado correctamente, debemos asegurarnos de que el peso *top* siempre sea mayor que la suma de los pesos de todas las cláusulas soft.

```
c
c comments Partial MaxSAT
c
p wcnf 3 4 10
10 -1 -2 0
1 -1 2 -3 0
1 -3 2 0
1 1 3 0
```

- Problema Weighted Partial MaxSAT

  - La línea de parámetros de este problema es la misma que en el problema Partial MaxSAT, es decir *p wcnf nbvar nbclauses top*. La única diferencia con este último es que en este caso, las cláusulas soft pueden tener distintos pesos, siempre y cuando estos sean menores a *top*. En este problema debemos asegurarnos también de que la suma de todos los pesos de las cláusulas soft sea menor que *top*.

```
c
c comments Weighted Partial MaxSAT
c
p wcnf 3 4 20
20 -1 -2 0
5 -1 2 -3 0
8 -3 2 0
6 1 3 0
```

Salida del resolutor

Los solvers deben responder con mensajes para que el usuario pueda interpretar y conocer la respuesta. El formato de salida está inspirado en el formato DIMACS. Este es un ejemplo de una respuesta correcta:

```
c
c Weighted MaxSAT Solver
c
```

```
o 521
o 434
o 237
s OPTIMUM FOUND
v -1 -2 3 -4 5 -6 7 -8 -9
```

Los mensajes pueden contener distintos bloques. Son los siguientes:

- Comentarios
  - Los comentarios empiezan siempre con el carácter *'c'* seguido de un espacio.
  - Son líneas opciones, y no tienen un orden o sitio establecido en la salida del resolutor.
  - Suelen contener información que el propio autor del solver quiere que el usuario conozca.
  - En una ejecución masiva de pruebas, se aconseja evitar estas lineas ya que no aportan ningún tipo de información relativa a las respuestas.
- Solución óptima actual
  - Estas líneas empiezan siempre con el carácter *'o'* seguido de un espacio.
  - Son líneas obligatorias.
  - Seguidamente, siempre encontraremos un entero que representa la mejor solución encontrada hasta el momento. Para un problema Weighted Partial MaxSAT este entero representaría la suma de los pesos de las cláusulas insatisfechas. En cambio, para un problema MaxSAT, representaría el número de cláusulas insatisfechas por la solución actual. Cada vez que se encuentra una nueva solución, el solver escribe una nueva línea de este tipo. El solver tomará siempre como solución óptima la última línea de este bloque.
- Respuesta
  - Esta línea siempre comenzará con el carácter *'s'* seguido de un espacio.
  - Es una línea obligatoria a la vez que única.
  - Nos indica cual es la respuesta al problema, y debe estar formada siempre por uno de estos valores:
    - **OPTIMUM FOUND**: este valor debe utilizarse cuando la última linea de tipo *'o'* sea la solución óptima del problema al dar esta una asignación completa de las variables de la fórmula.

- ◇ **UNSATISFIABLE**: este valor significa que no se han podido cumplir todas las cláusulas hard del problema.
- ◇ **UNKNOWN**: en cualquier otro caso, se utilizará este valor, por ejemplo, para indicar que se ha encontrado una solución pero que no es la óptima.

- Asignación de valores

  - Estas líneas empezarán con el carácter *'v'* seguido de un espacio.
  - Se permite más de una línea de este tipo, pero en ese caso, debe considerarse el resultado al juntar todas las líneas.
  - Si el resolutor ha encontrado una solución óptima, debe mostrar en este bloque, una asignación de verdad para las variables que conforman el problema. Dicho de otra forma, debe proporcionar un *listado de literales* no complementarios que al ser interpretados a cierto nieguen el mínimo número de cláusulas soft en un problema MaxSAT o minimicen la suma de los pesos de las cláusulas insatisfechas para un problema Weighted MaxSAT.
  - Un *literal* se representa con un número entero que identifica a la variable. Por otra parte, la negación de una variable, se denota con el símbolo *menos(-),* seguido del entero al cual representa la variable.
  - Una línea de este tipo debe contener un valor para todas las variables, sin importar el orden de los literales.
  - Si el resolutor no escribe una línea de asignación de valores, se considerará que la respuesta es de tipo UNKNOWN.

Capítulo 3

Codificación del TSP en MaxSAT

En este capítulo veremos cómo codificar instancias TSP cómo un problema MaxSAT, concretamente en una codificación Weighted Partial MaxSAT, la cual es una generalización del problema MaxSAT tal y como hemos podido observar en el capítulo 2.4. La codificación en Weighted Partial MaxSAT consta de dos partes, la parte hard y la parte soft. En la parte hard, se encuentran las cláusulas del problema que deben ser obligatoriamente satisfechas, mientras que en la parte soft, encontramos las cláusulas que pueden o no ser satisfechas, ya que éstas representan en esencia la función objetivo del problema de optimización.

En el caso concreto del TSP, en la parte hard estarán las cláusulas que se encargarán de asegurar que las ciudades forman un camino hamiltoniano, es decir, un camino que tiene que pasar por todas las ciudades exactamente una vez, comenzando y terminando en la misma ciudad. Las cláusulas de la parte soft del problema representan las acciones de viajar de una ciudad a otra con su coste asociado. En función de que cláusulas soft se satisfagan, describiremos un determinado trayecto y su coste asociado.

Representamos nuestro mapa de ciudades mediante un grafo dirigido con pesos en los arcos, $G = (N, A)$, en donde N es el conjunto de nodos tal que $n_i \epsilon V$ representa la ciudad iésima y A es el conjunto de arcos tal que $(i, j) \epsilon E$ representa que las ciudades iésima y jésima están conectadas.

A continuación, describimos el conjunto de variables y restricciones (cláusulas) que nos permiten codificar una instancia del TSP como una instancia del problema MaxSAT.

Disponemos de un conjunto de variables $b_i^j \epsilon B$, tal que b_i^j evalua a cierto si la ciudad iésima ocupa la posición jésima en la solución al TSP.

Dividimos el conjunto de cláusulas, entre hard y soft. Para simplificar nuestra notación utilizaremos expresiones del tipo $CNF(\sum_{i=1}^{n} b_i = 1)$ que representa la restricción *exactly_ one* sobre un conjunto de variables booleanas $(b_1, ..., b_n)$. Dicha restricción evalúa a cierto, si y sólo si, se evalúa a cierto exactamente una de las variables booleanas. Traducimos esta restricción mediante el siguiente conjunto de cláusulas:

$$(b_1 \vee ... \vee b_n) \wedge \cup_{i=1,j=1,i\neq j}^{n} (\neg b_i \vee \neg b_j)$$

Restricciones Hard:

Nuestra solución debe corresponer a un ciclo hamiltoniano en el grafo G del TSP. Utilizamos las siguientes cláusulas:

- H1: cada una de las ciudades debe ser visitada exactamente una vez

$$\cup_{i=1}^{n} CNF(\sum_{j=1}^{n} b_i^j = 1)$$

- H2: en cada posición del trayecto se visitará exactamente una ciudad

$$\cup_{j=1}^{n} CNF(\sum_{i=1}^{n} b_i^j = 1)$$

- H3: en el caso de que el grafo no fuese completo, es decir, algunas transiciones entre ciudades no están permitidas, debemos añadir las siguientes cláusulas:

$$\cup_{(u,v)\notin E} \ \cup_{j=1}^{n} (\neg b_u^j \vee \neg b_v^{j+1 mod(n)})$$

Las restricciones H1 y H2 se corresponden con la restricción *alldiff (*All different constraint), en donde dado un grupo de variables, se obliga a que cada variable de dicho grupo asuma un valor diferente al valor de todas las demás variables del grupo. Dicho de otra forma, no habrá dos variables con el mismo valor.

Restricciones Soft:

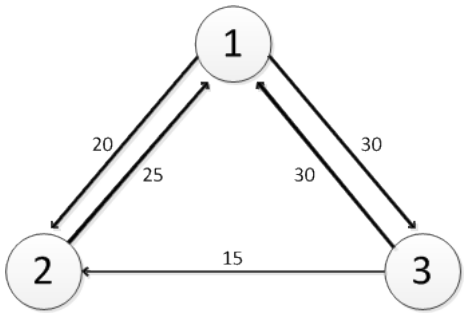

Figura 3.0.1: Grafo que representa una instancia del TSP

Las cláusulas de la parte soft hacen referencia a la optimización del problema, que en nuestro caso concreto significa encontrar un trayecto con coste óptimo, el cual se obtiene de sumar los pesos de los arcos que conforman el trayecto. En la parte soft se deben representar todos los arcos con su coste asociado. Para tal efecto, utilizamos las siguientes cláusulas:

$$\cup_{(u,v)\epsilon E} \ \cup_{j=1}^{n} \omega(u,v) : (\neg b_u^j \vee \neg b_v^{j+1mod(n)})$$

en donde $\omega(u,v)$ es el peso del arco(u,v).

Puesto que hemos definido el problema MaxSAT como la minimización de la suma de los pesos de las cláusulas falsificadas, debemos modelizar que cuando un arco forma parte de la solución se debe falsificar una cláusula cuyo peso es el peso del arco.

Ejemplo

Imaginemos que disponemos del siguiente grafo (figura 3.0.1), el cual representa una instancia del TSP y está formado por 3 nodos que hacen referencia a las ciudades 1, 2 y 3.

Siguiendo la nomenclatura mencionada anteriormente, las variables de las que disponemos son las siguientes:

- b_1^1, b_1^2, b_1^3, b_2^1, b_2^2, b_2^3, b_3^1, b_3^2, b_3^3

Llegados a este punto, es momento de introducir las cláusulas que formarán la parte hard del problema, siguiendo con las restricciones explicadas en la codificación en MaxSAT:

- Cláusulas resultantes al aplicar la restricción H1 para la ciudad 1:
  - $(b_1^1 \vee b_1^2 \vee b_1^3) \wedge (\neg b_1^1 \vee \neg b_1^2) \wedge (\neg b_1^1 \vee \neg b_1^3) \wedge (\neg b_1^2 \vee \neg b_1^3)$
- Cláusulas resultantes al aplicar la restricción H2 para la posición 1 del trayecto:
  - $(b_1^1 \vee b_2^1 \vee b_3^1) \wedge (\neg b_1^1 \vee \neg b_2^1) \wedge (\neg b_1^1 \vee \neg b_3^1) \wedge (\neg b_2^1 \vee \neg b_3^1)$
- En este ejemplo disponemos de un grafo el cual no dispone del arco (2,3), es decir, en nuestra instancia del TSP no disponemos de un camino que vaya desde la ciudad 2 hacia la ciudad 3. Las cláusulas que debemos introducir, son las siguientes:
  - $(\neg b_2^1 \vee \neg b_3^2) \wedge (\neg b_2^2 \vee \neg b_3^3) \wedge (\neg b_2^3 \vee \neg b_3^1)$

Vamos ahora a ver que cláusulas representan la parte soft, aplicando las restricciones descritas anteriormente para la parte soft:

- Cláusulas resultantes al aplicar las restricciones soft para el arco que va desde la ciudad 1 hacia la ciudad 2:
  - $(\neg b_1^1 \vee \neg b_2^2) \wedge (\neg b_1^2 \vee \neg b_2^3) \wedge (\neg b_1^3 \vee \neg b_2^1)$

Tal y como hemos mencionado anteriormente, todas las cláusulas de la parte soft están asociadas a un peso determinado.

Finalmente, veremos cual es el fichero resultante de aplicar todas las restricciones, en donde las variables b_i^j se representan siguiendo la expresión $b_i^j \rightarrow j + (i-1) * n$, y de posteriormente codificar las cláusulas en formato Weighted Partial MaxSAT (sección 4.3).

```
p wcnf 9 42 121
c <PARTE HARD>
c restricción H1
121 1 2 3 0
121 -1 -2 0
121 -1 -3 0
121 -2 -3 0
121 4 5 6 0
121 -4 -5 0
121 -4 -6 0
121 -5 -6 0
121 7 8 9 0
```

```
121 -7 -8 0
121 -7 -9 0
121 -8 -9 0
c restricción H2
121 1 4 7 0
121 -1 -4 0
121 -1 -7 0
121 -4 -7 0
121 2 5 8 0
121 -2 -5 0
121 -2 -8 0
121 -5 -8 0
121 3 6 9 0
121 -3 -6 0
121 -3 -9 0
121 -6 -9 0
c restricción H3
121 -4 -8 0
121 -5 -9 0
121 -6 -7 0
c <PARTE SOFT>
20 -1 -5 0
30 -1 -8 0
25 -4 -2 0
30 -7 -2 0
15 -7 -5 0
20 -2 -6 0
30 -2 -9 0
25 -5 -3 0
30 -8 -3 0
15 -8 -6 0
20 -3 -4 0
30 -3 -7 0
25 -6 -1 0
30 -9 -1 0
15 -9 -4 0
```

Capítulo 4

Arquitectura e implementación

En este capítulo se explica cómo se ha trabajado en la implementación del entorno web, así como las distintas funcionalidades que éste puede aportar al usuario.

Concretamente, se describe el módulo que se ha desarrollado para la correcta interacción con la interfaz proporcionada por Google Maps, así como la implementación sobre PHP que se ha utilizado del protocolo XML-RPC.

Cabe destacar que para el correcto desarrollo y óptimo funcionamiento de esta aplicación se han utilizado distintas tecnologías y lenguajes de programación, tales como: JavaScript, PHP, AJAX, XML-RPC, HTML y CSS.

La figura 4.0.1 nos muestra cual es la estructura del entorno web.

En el último punto del capítulo se habla de como se ha llevado a cabo la resolución del problema, utilizando para ello la codificación descrita en el capítulo 3.

4.1. Definición y visualización del problema

El módulo TSPLib, implementado en *JavaScript*, es el responsable de que la interacción entre el usuario y los servicios proporcionados por *Google*, sea posible. Además, también se encarga de enviar los datos introducidos por el usuario al

Figura 4.0.1: Estructura del entorno web

servidor XML-RPC, el cual devolverá una solución a la instancia TSP, formulada por el usuario. Una vez recibida la solución, este módulo nos proporcionará una visualización gráfica de la ruta, utilizando para ello, la interfaz que nos proporciona Google Maps. Es importante mencionar que el API de Google Maps añade alguna que otra restricción a esta aplicación, como por ejemplo, el tiempo de espera entre las diferentes peticiones que podemos hacer a los servicios que éste nos proporciona, por lo que el tiempo de espera por parte del usuario, puede verse incrementado.

Hay 2 conceptos que deben tenerse en cuenta a la hora de utilizar este módulo. Son los siguientes:

- ***Geocodificación***: la geocodificación es el proceso que transforma una dirección postal, en unas coordenadas geográficas. Estas coordenadas, pueden ser utilizadas luego para localizar un punto en un mapa real.
- ***Geocodificación inversa***: la geocodificación inversa, como bien su nombre indica, es el proceso de asignación de una dirección postal a unas coordenadas geográficas. Este concepto es un componente básico de los sistemas basados en localización, ya que convierte una coordenada en una dirección postal, lo que es más fácil de entender por el usuario final.

A continuación se detallarán cada una de las funciones implementadas en este módulo, así como los atributos utilizados por la misma.

Los atributos que conforman este módulo son los siguientes:

- ***map***: esta variable representa al mapa proporcionado por *Google Maps*, que podemos visualizar por pantalla.

Figura 4.1.1: Panel contenedor de la información relativa a la ruta

- ***markersArray***: en este vector guardaremos todos los puntos elegidos por el usuario, los cuales posteriormente, se utilizarán para buscar la ruta óptima.
- ***routes***: esta variable contiene las distancias entre cada par de puntos del vector *markersArray*. Esta cadena es la que enviaremos al servidor *XML-RPC* para que este disponga de toda la información necesaria para resolver el problema.
- ***directionsDisplay***: gracias a este objeto, proporcionado también por *Google Maps*, podremos visualizar la ruta en el mapa, una vez el servidor *XML-RPC* haya encontrado la solución.
- ***summaryPannel***: esta variable representa el panel de la página web en donde mostraremos toda la información relativa a la ruta obtenida, tal y como se puede observar en la figura 4.1.1.
- ***statusPannel***: al igual que el objeto *summaryPannel*, esta variable representa un espacio de la página web, pero en este caso, reservado a mostrar

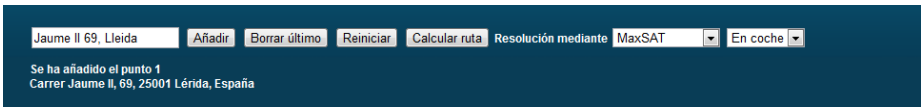

Figura 4.1.2: Panel de estado

información en todo momento de los sucesos que puedan ocurrir en el mapa, tal y como se puede observar en la figura 4.1.2

- ***bounds***: este objeto es el responsable de centrar y ajustar el mapa, para que en todo momento podamos ver todos los puntos que hayamos elegido.
- ***infoWindow***: mediante este objeto, cada vez que el usuario añade un nuevo punto en el mapa, o hace *click* en uno ya presente en él, podemos visualizar una ventana que contiene información relativa a ese punto.
- ***travelMode***: tal y como su nombre indica, esta variable contiene el modo de viaje que el usuario ha elegido para su ruta.
- ***solver***: esta variable nos indica qué resolutor debe ejecutarse en el servidor para resolver el problema.
- ***totalDistance***: esta variable contiene la distancia total, del trayecto, en kilómetros.
- ***totalTime***: esta variable contiene el tiempo total estimado en recorrer el trayecto.

Las funciones implementadas en el módulo TSPLib son estas:

- ***initialize***: como su nombre indica, está función se encarga de inicializar toda la aplicación, para que el usuario pueda formular correctamente su problema. Además, mediante la función *geoLocation*, es capaz de detectar la ubicación actual del usuario, para centrar el mapa en ella, siempre que el usuario así lo desee. Esta función también se encarga de añadir un evento, para que cuando el usuario haga *click* en el mapa, se añada un nuevo punto al mapa mediante la función *addMarkerByClick*. En la figura 4.1.3 podemos observar que aspecto tiene la aplicación en un primer instante.
- ***geolocation***: esta función es capaz de detectar la ubicación del usuario con el objetivo de centrar el mapa en ella, siempre y cuando, obviamente, el usuario lo permita. Si no es posible encontrar la ubicación del usuario o este no ha permitido dicha acción, la función *handleNoGeolocation* se encargará de centrar el mapa en un punto predeterminado.

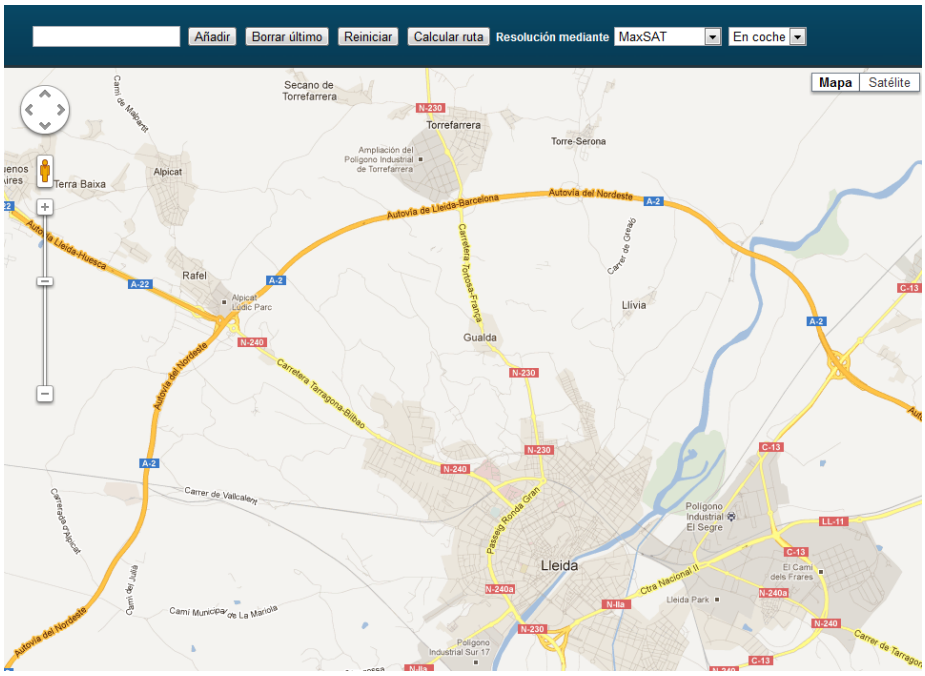

Figura 4.1.3: Inicialización del mapa

- ***handleNoGeolocation***: si la aplicación no ha sido capaz de detectar la ubicación del usuario, o este no ha permitido ejecutar dicha acción, esta función centrará el mapa en un punto predeterminado del mapa. Esto es necesario, ya que se requiere que el mapa se inicialice en un punto en concreto.

- ***addMarkerByClick***: este método añade un nuevo punto al mapa, cuando se detecta un *click* del usuario, utilizando para ello la función *addMarker*. Esto es posible gracias a la geocodificación inversa, la cual a partir de unas coordenadas, es capaz de transformar dichas coordenadas en una dirección postal, que obviamente, es mucho más fácil de entender e interpretar por el usuario final.

- ***addMarkerByGeocoding***: al tiempo que la función *addMarkerByClick* utiliza la geocodificación inversa, esta utiliza la geocodificación con el objetivo de transformar una dirección postal, que el usuario ha indicado, en coordenadas, para que la aplicación sea capaz de interpretar estas últimas para añadir un punto al mapa mediante la función *addMarker*. Si el usuario no ha sido lo bastante específico con la descripción de la dirección, correspondiéndose esta con múltiples coordenadas, se le pedirá que ajuste más su búsqueda con el objetivo de añadir correctamente el punto que nos pide. La figura 4.1.4 nos muestra esta situación.

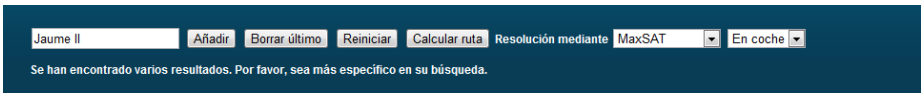

Figura 4.1.4: Mensaje provocado al introducir una dirección que se corresponde con múltiples resultados

- ***addMarker***: esta es la función que se encarga de añadir los puntos geográficos que posteriormente conformarán la ruta. Para ello, se utilizan unos parámetros de entrada, los cuales nos indican las coordenadas y la descripción del nuevo punto a añadir. Gracias al vector *markersArray* podemos saber que índice le corresponde al nuevo punto, lo cual es necesario para elegir el icono que le corresponde. Una vez el punto está creado y con el objetivo de que el usuario pueda ver en cualquier momento la información asociada a este, también introducimos una ventana de información (objeto *infoWindow*) que es visible cada vez que el usuario hace *click* en un punto. Finalmente, este método decide si debe ajustarse o no el *zoom* del mapa para que el usuario pueda ver todos los puntos que ha ido introduciendo. La figura 4.1.5 nos muestra como queda el mapa después de añadir un nuevo punto.

- ***clearOverlays***: utilizamos esta función para limpiar cualquier elemento visual que contenga el mapa, ya sean puntos, rutas o ventanas de información. No obstante, no es suficiente con solo eliminar los objetos visuales, ya que detrás de estos, encontramos siempre un objeto que los representa, por lo que también es preciso tratar estos objetos. Por ejemplo, cuando eliminamos todos los puntos del mapa, debemos asegurarnos de que el vector que contiene los puntos, *markersArray*, quede completamente limpio, es decir, sin ningún elemento. Lo mismo pasa con el objeto que representa la ruta, *directionsDisplay*.

- ***deleteLast***: esta función permite al usuario eliminar el último punto que se ha introducido en el mapa, modificando para ello el objeto *markersArray*. Si en el momento de eliminar un punto, hay una ruta propuesta en el mapa, está será eliminada, ya que obviamente, al eliminar un punto, la ruta debe ser otra, por lo que el proceso deberá ser lanzado de nuevo.

- ***calcRoute***: este método se encarga de calcular la distancia que hay entre cada par de puntos, los cuales están almacenados en el vector *markersArray*. Para ello, utilizamos un servicio que nos proporciona el API de Google Maps, el cual dados dos puntos, nos devuelve la distancia entre estos, teniendo en cuenta las opciones que haya podido elegir el usuario, como por ejemplo, el modo de viaje. Debemos considerar que la distancia entre dos puntos no tiene por qué ser la misma en los dos sentidos, por eso,

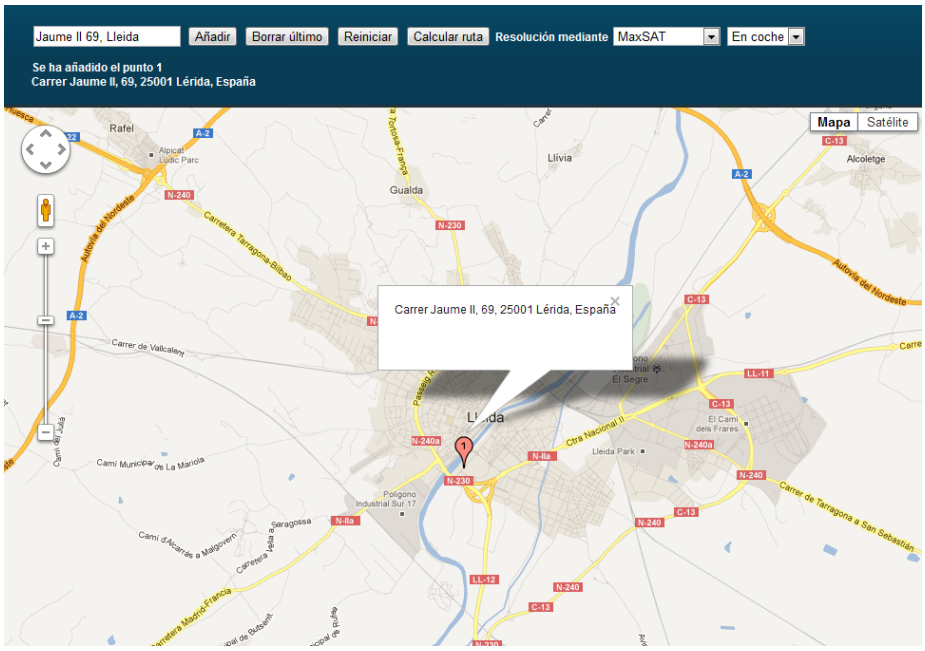

Figura 4.1.5: Añadir nuevo punto al mapa

por cada par de puntos, debemos hacer 2 peticiones a dicho servicio. Una vez finalizado el proceso, guardaremos toda la información en la variable *routes*, y se llamará a la función *sendRoute*, con el objetivo de establecer una comunicación con el servidor XML-RPC. Cabe destacar que la ubicación de los puntos introducidos por el usuario puede variar automáticamente una vez construida la ruta. Eso se debe a que el usuario puede haber insertado un punto en un sitio donde, por ejemplo, sea imposible la circulación con coche. No obstante, esta variación en la mayoría de los casos será mínima. La figura 4.1.6 nos muestra el diagrama de secuencia UML de la función *calcRoute*.

- ***sendRoute***: esta función se encarga de establecer la comunicación con el servidor XML-RPC, mandándole a este toda la información necesaria para resolver el problema, es decir, los puntos y la distancia entre cada par de estos, la cual tenemos almacenada en la variable *routes*. Una vez recibida la solución por parte del servidor, llamaremos a la función *showRoute* para que esta interprete la solución que el servidor nos ha mandado, y muestre por pantalla la ruta obtenida.
- ***showRoute***: tal y como su nombre indica, esta función es la responsable de que podamos ver el resultado final, el cual nos ha mandado el servidor XML-RPC, transformando dicho resultado en una ruta que podremos vi-

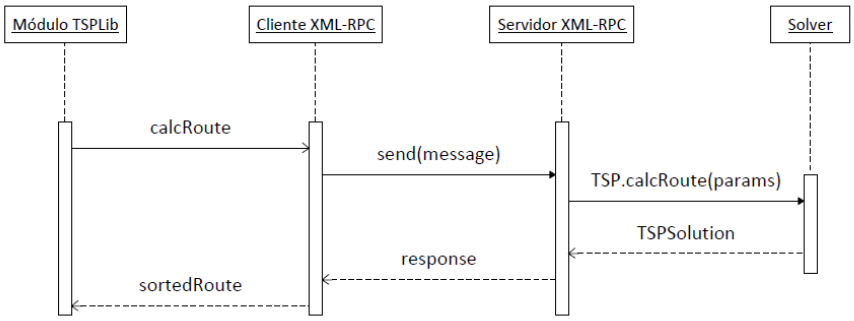

Figura 4.1.6: Diagrama de secuencia UML de la función *calcRoute*

sualizar en la interfaz proporcionada por Google. Para construir la ruta correctamente, necesitamos modificar los iconos de los puntos situados en el mapa, para que estos queden en el orden correcto, es decir, en el orden óptimo para que la ruta tenga un coste mínimo. Una vez tenemos esto, simplemente necesitamos añadir una línea que conecte todos estos puntos, respetando siempre las opciones que el usuario pueda haber elegido previamente, como puede ser el modo de viaje.Además, en el panel de estado, podremos encontrar la distancia total de la ruta, así como el tiempo estimado empleado en recorrerla. La figura 4.1.7 nos muestra como el usuario puede visualizar por pantalla la ruta óptima formada por los puntos que él mismo ha elegido previamente.

4.2. Comunicación mediante XML-RPC

El motivo por el cual se decidió utilizar el protocolo XML-RPC es que se quiere proporcionar una arquitectura capaz de intercambiar y extender los servicios de visualización y resolución del TSP. Por lo tanto, el entorno web necesitará implementar un cliente XML-RPC en la parte de definición y visualización del problema, y un servidor XML-RPC en la parte de resolución del mismo.

Para implementar el cliente y el servidor XML-RPC, se ha utilizado la librería *XML-RPC for PHP*, disponible en `http://phpxmlrpc.sourceforge.net`

El servidor XML-RPC actúa como intermediario entre los servicios de definición y visualización del problema y la parte de resolución del mismo. Es por esto, que es un componente básico en esta aplicación, además de reutilizable por otras aplicaciones, tal y como ya hemos podido observar en la sección 2.3.

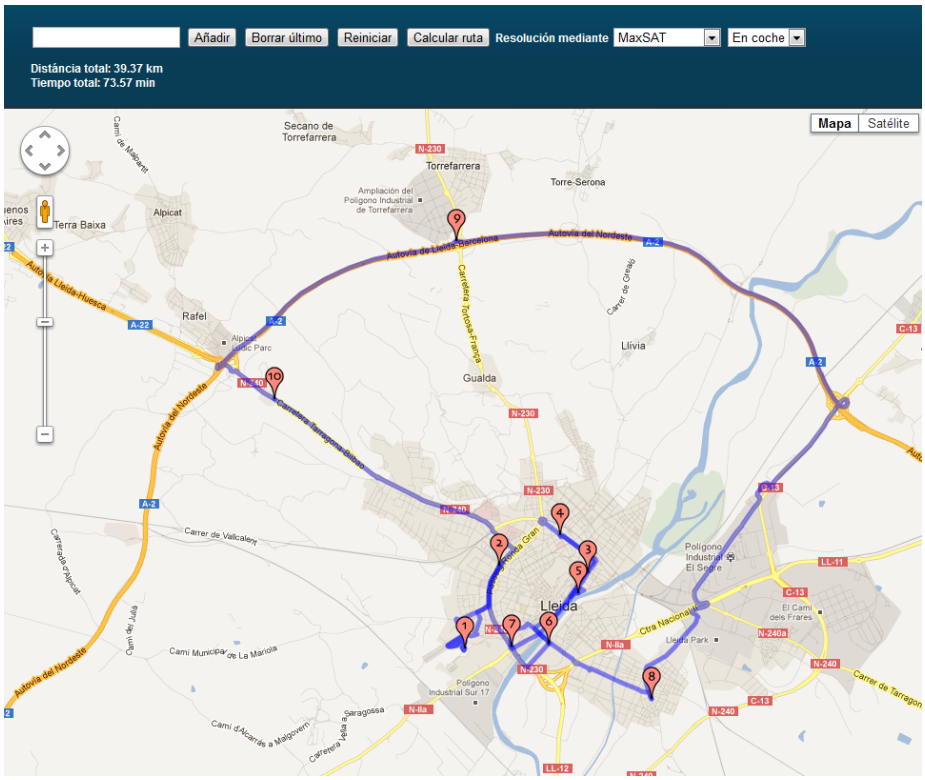

Figura 4.1.7: Ruta óptima que se obtiene al resolver el TSP

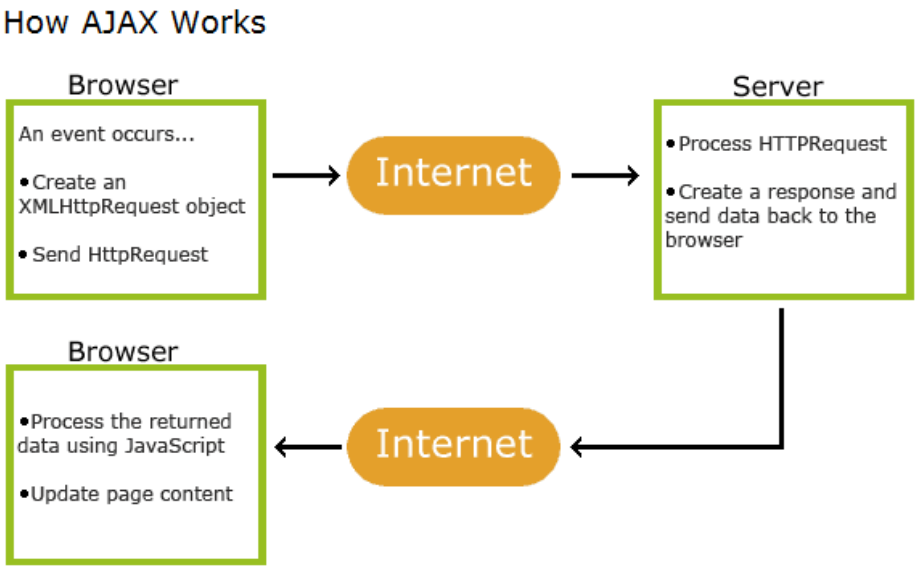

Figura 4.2.1: Funcionamiento de AJAX

El cliente XML-RPC, por su parte, es el responsable de mandar un mensaje al servidor XML-RPC solicitando una solución a la instancia del TSP formulada por el usuario mediante la interfaz de Google Maps. Esta petición podría haberse llevado a cabo desde el módulo TSPLib, implementado en JavaScript, pero se prefirió hacerlo así, para que el cliente y el servidor XML-RPC utilizaran la misma implementación del protocolo XML-RPC, es decir, la librería *XML-RPC for PHP*. Una vez recibida la solución por parte del servidor, ésta se enviará al módulo TSPLib, con el objetivo de que éste interprete la solución y el usuario pueda visualizar la ruta por pantalla.

4.2.1. Cliente XML-RPC

El cliente XML-RPC recibe los datos necesarios para resolver el problema desde el módulo TSPLib, mediante *AJAX* (Asynchronous JavaScript and XML). AJAX nos permite intercambiar datos con un servidor y actualizar partes de una web sin necesidad de recargar toda la página, tal y como se puede observar en la figura 4.2.1.

Para ello, se ha utilizado el método POST, el cual nos permite recibir una gran cantidad de datos. En este caso, estos datos son el número de puntos que el usuario ha introducido, además de todas las distancias entre cada par de estos puntos y el resolutor que debe utilizarse. Una vez disponemos de toda esta

información, el cliente XML-RPC, manda un mensaje al servidor XML-RPC, esperando recibir una respuesta con la solución al problema. Este mensaje debe enviarse con unos parámetros adecuados, los cuales podremos ver en detalle en la siguiente sección.

La petición que el cliente realiza al servidor es la siguiente:

```
$server = new xmlrpc_client('Server.php')
$message = new xmlrpcmsg(TSP.calcRoute', array($distances));
$message ->addParam(new xmlrpcval($nodes, 'int'));
$message ->addParam(new xmlrpcval($solver, 'string'));
$result = $server->send($message);
```

en donde *TSP.calcRoute* es la función que debe ejecutarse en el servidor, *$distances* es una array que almacena la distancia entre todos los puntos, *$nodes* contiene el número de puntos que va a contener la ruta y *$solver* indica qué resolutor será el encargado de resolver el problema.

Una vez se ha recibido la respuesta por parte del servidor, el cliente XML-RPC, enviará la solución al módulo TSPLib, para que éste pueda interpretar la solución y mostrar la ruta por pantalla.

4.2.2. Servidor XML-RPC

Este servidor solo dispone de una función, la cual lleva por nombre *TSP.calcRoute*. Obviamente, esta se ejecutará cuando el servidor reciba un mensaje en el que se indique que se desea utilizar dicha función. Como se ha visto en la sección anterior, el mensaje debe enviarse con los parámetros adecuados. Todas las funciones de un servidor XML-RPC deben tener una firma, o dicho de otra forma, una definición en donde se indiquen los parámetros que deben enviarse junto al mensaje, cuando se desee utilizar la función en cuestión.

```
$calcRoute_sig = array(array('struct', $xmlrpcValue,
                                       $xmlrpcInt,
                                       $xmlrpcString));
```

En el caso de la función *TSP.calcRoute*, los parámetros que se deben enviar son los siguientes:

- ***distances***: este parámetro, el cual es una array, debe almacenar las distancias entre todos los puntos que conformarán la ruta.

- ***nodes***: este parámetro es un entero el cual contiene el número de puntos de que dispondrá la ruta.
- ***solver***: este parametro es una cadena que nos indica qué resolutor debe utilizarse para resolver el problema

Una vez disponemos de toda la información, el servidor escribirá un fichero con la siguiente estructura, el cual es necesario para poder codificar correctamente el problema en MaxSAT:

```
3 6
1 2 3.711
1 3 7.639
2 1 3.627
2 3 4.052
3 1 7.41
3 2 4.281
```

Este fichero representa un grafo completo de 3 nodos. La primera línea del fichero nos indica el número de nodos y el número de aristas del grafo, respectivamente. Las demás líneas contienen la distancia o peso entre cada par de nodos del grafo completo.

Este fichero es necesario para llamar al script *runSolver*, el cual ejecuta las siguientes acciones:

- Codificar el TSP según el resolutor que se haya elegido
- Resolver el problema mediante el resolutor seleccionado
- Devolver el resultado, es decir, el orden en que se deben visitar las ciudades

Una vez llegados a este punto, el servidor XML-RPC ya puede mandar una respuesta, que en este caso contendrá la solución al TSP, al cliente XML-RPC del cual ha recibido la petición.

```
return new xmlrpcesp(new xmlrpcval(exec('./runSolver.sh'
                                        . ' ' .
                                        $nodes
                                        . ' ' .
                                        $solver), 'string'));
```

Esta respuesta está formada por una cadena que nos indica en que posición deben recorrerse los puntos que conformarán la ruta. Será posteriormente el cliente, quien interprete esta respuesta para mostrar la ruta definitiva por pantalla.

4.3. Resolución del problema

En el capítulo 3 hemos visto como codificar una instancia del TSP en un problema MaxSAT. No obstante, el resolutor necesita leer el problema con un formato en concreto (sección 2.4).

Por este motivo, se necesita una herramienta, en este caso la clase ***graph***, que se encargue de transformar las cláusulas resultantes de aplicar las restricciones de la sección 3 para una instancia TSP, en un fichero con formato Weighted Partial MaxSAT.

A continuación, se describen los atributos y funciones de los que dispone la clase *graph*.

Atributos

- ***nEstados***: número de ciudades
- ***nTransiciones***: número de arcos
- ***nClauses***: número de cláusulas
- ***nVariables***: número de variables
- ***transiciones***: matriz que almacena los pesos de los arcos
- ***topWeight***: suma de todos los pesos de los arcos

Funciones

- ***inicializar_problema***: esta función abre el fichero que describe una instancia del TSP e inicializa los atributos mencionados anteriormente. También saca por la salida estándar la cadena *p wcnf nVariables nClauses,* que es la primera línea que debe aparecer en un fichero con formato Weighted Partial MaxSAT.
- ***restriccion_filas***: esta función se encarga de aplicar la restricción H1 (capítulo 3), utilizando para ello la restricción *exactly_one*, para posteriormente transformar las cláusulas resultantes en formato Dimacs. El resultado obtenido se saca por la salida estándar.
- ***restriccion_columnas***: realiza las mismas tareas que la función *restriccion_filas* pero aplicando ahora la restricción H2 (capítulo 3).
- ***restriccion_hamiltoniana***: esta función se encarga de aplicar la restricción H3 (capítulo 3), para posteriormente transformar las cláusulas resultantes en formato Dimacs. El resultado obtenido se saca por la salida estándar.

- ***restriccion_soft***: esta función se encarga de aplicar la restricción Soft (capítulo 3), para posteriormente transformar las cláusulas resultantes en formato Dimacs. El resultado obtenido se saca por la salida estándar.
- ***displayBoard:*** esta función se encarga de interpretar la solución devuelta por un resolutor MaxSAT, para posteriormente informar al usuario del orden en que deben visitarse las ciudades.

La clase graph necesita recibir por parámetro un fichero que represente a un grafo. Siguiendo el ejemplo de la sección 3, el fichero tendría el siguiente aspecto:

```
3 5
1 2 20
1 3 30
2 1 25
3 1 30
3 2 15
```

donde la primera línea del fichero nos indica el número de nodos y el número de arcos del grafo, respectivamente. Las demás líneas, contienen el peso que hay entre cada par de nodos, donde los primeros dos enteros representan a los nodos y el tercer valor representa el peso.

Una vez recibido el fichero, se llamarán a las funciones *inicializar_problema*, *restriccion_filas*, *restriccion_columnas* y *restriccion_hamiltoniana*, para que éstas transformen las cláusulas, resultantes de aplicar las correspondientes restricciones descritas en el capítulo 3, en formato Weighted Partial MaxSAT.

Esta clase no sólo disponde de la funcionalidad que le permite transformar una instancia del TSP en formato Weighted Partial MaxSAT. Además de esto, una vez disponemos de la solución que nos devuelven los resolutores MaxSAT (sección 2.4), la función *displayBoard* es capaz de interpretar esta respuesta, con el objetivo de generar una nueva solución que sea más facil de interpretar por parte del usuario, indicando en que orden deben visitarse las ciudades.

Capítulo 5

Resultados experimentales

En este capítulo se muestran los resultados experimentales que se han obtenido al resolver algunas instancias del TSP, utilizando para ello el resolutor WMax-Satz.

Además, también se comparan los resultados obtenidos, con otras aplicaciones web similares a esta.

Para llevar a cabo los experimentos, se ha utilizado una máquina con las siguientes especificaciones:

- Procesador: Intel Core i3 370M / 2.4Ghz (Dual Core)
- Memoria RAM: 4 GB
- Memoria Caché: 3 MB
- Sistema operativo: Ubuntu 12.04

Las 14 ciudades que se han utilizado para definir las instancias del TSP son: Lleida, Zaragoza, Logroño, Valladolid, Madrid, Salamanca, León, Vigo, Santander, Oviedo, Valencia, Córdoba, Albacete, y Badajoz, tal y como se puede observar en la figura 5.0.1.

Los resultados de la siguiente tabla nos indican cuanto tiempo han empleado los resolutores en resolver el problema así como la distancia óptima del trayecto que estos han obtenido.

Figura 5.0.1: Ciudades utilizadas en las instancias de prueba del TSP

Resolutor	WMaxSatz	
	Tiempo	Distancia óptima
10 ciudades	22”	2.472 km
11 ciudades	2’ 30”	2.718 km
12 ciudades	24’ 45”	3.285 km
13 ciudades	125’ 50”	3.338 km
14 ciudades	746’ 30”	3.563 km

Las conclusiones que podemos sacar de estos resultados es qué el resolutor WMaxSatz obtiene mejor tiempo de respuesta cuando el número de ciudades no es muy elevado, ya que cuando éste se ve incrementado, los tiempos de resolución obtenidos se ven incrementados notablemente.

A continuación, se comparan las distancias óptimas que han obtenido los resolutores utilizados en este proyecto con las distancias obtenidas, utilizando obviamente las mismas ciudades, por otras aplicaciones web, tales como:

- *RouteXL*, disponible en http://www.routexl.com/
- *OptiMap*, disponible en http://gebweb.net/optimap/
- *FindTheBestRoute*, disponible en http://findthebestroute.com/RouteFinder.html

	Aplicación		
	RouteXL	OptiMap	FindTheBestRoute
10 ciudades	2.486 km	2.478 km	2.478 km
11 ciudades	2.724 km	2.720 km	2.839 km
12 ciudades	3.291 km	3.286 km	3.464 km
13 ciudades	3.349 km	3.343 km	3.902 km
14 ciudades	3.608 km	3.584 km	4.162 km

Después de analizar los resultados de la comparativa anterior, se puede decir que los resultados obtenidos por el resolutor WMaxSatz son muy parecidos a los obtenidos por las aplicaciones *RouteXL* y *OptiMap*, mientras que la aplicación *FindTheBestRoute*, a medida que va aumentando el número de ciudades, va obteniendo peores resultados, por lo que podemos concluir que la aplicación presentada en este proyecto, está a la altura de otras aplicaciones ya consolidadas en la web, por lo que a resultados se refiere, ya que el tiempo de espera es mucho mayor por nuestra parte.

Cabe destacar que Google Maps, en su página web https://maps.google.com/ no proporciona un servicio de optimización de rutas. La razón por la cual Google no proporciona este servicio es qué éste a su vez es una variante del TSP, el cual es un problema NP-Completo. Si Google Maps resolviera el TSP, obviamente utilizaría medios computacionales mucho más potentes a los cuales muy poca gente tiene acceso, por lo que posiblemente, mucha gente aprovecharía este servicio para resolver problemas NP-Completos, provocando que otros miles de usuarios no pudieran utilizar Google Maps como han estado haciendo hasta ahora, ya que el servidor estaría colapsado intentado resolver los problemas NP-Completos.

Capítulo 6

Conclusiones y futuro trabajo

En este proyecto se ha conseguido separar la parte que nos permite resolver instancias del TSP, de la parte que permite definir y visualizar instancias reales del mismo, por lo que ésta última puede ser utilizada por otros usuarios, para que estos, posteriormente, puedan resolver mediante otras técnicas, el TSP u otros problemas que puedan ser definidos mediante un mapa.

Además, al estar utilizando el protocolo XML-RPC, cualquier usuario que desee resolver instancias del TSP, sin la necesidad de visualizar dichas instancias en un mapa, solo debe enviar una petición a nuestro servidor XML-RPC con el objetivo de que éste resuelva el problema, para que posteriormente el usuario pueda tratar la respuesta de la forma que más le convenga.

Un futuro proyecto sería trabajar con problemas con una mayor complejidad. Un problema que podría tratarse es el VRP (Vehicle Routing Problem), el cual es en realidad un amplio conjunto de variantes y personalizaciones de problemas, en los que se trata de dar servicio a unos clientes con una flota de vehiculos. Una variante muy conocida es el CVRP (Capacitated VRP), donde los vehículos tienen una capacidad limitada de la carga que deben entregar.

Por otra parte, sería interesante utilizar Open Street Maps en vez de Google Maps, ya que éste nos añade alguna que otra restricción, tal y como hemos visto en la sección 2.2.

Por lo que a la resolución del problema se refiere y después de analizar los resultados experimentales, podemos decir que este proyecto no propone una herramienta especialmente rápida para la resolución del TSP. Esto se debe a que hemos utilizado codificaciones que son mucho más eficientes y útiles en problemas con una mayor dificultad, tales como el VRP, mencionado anteriormente,

donde se dan otras restricciones adicionales que pueden ser tratadas eficientemente por SAT solvers. Por lo tanto, otro futuro proyecto sería utilizar otras técnicas de codificación para reformular el TSP así como la utilización de otros resolutores, para posteriormente poder contrastar los resultados.

Para finalizar, me gustaría destacar que este proyecto me ha sido de mucha utilidad para descubrir distintas tecnologías relacionadas con el mundo web tales como JavaScript, PHP, Python, XML-RPC, C++ y AJAX, a la vez que instructivo para conocer en mayor profundidad algunas de las técnicas utilizadas para resolver problemas de optimización combinatoria.

Bibliografía

[Ansotegui et al., 2009] Ansotegui, C., Bonet, M. L., and Levy, J. (2009). Solving (weighted) partial maxsat through satisfiability testing. In *Proc. of the 12th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT'09)*. 2.4

[Ansotegui et al., 2010] Ansotegui, C., Bonet, M. L., and Levy, J. (2010). A new algorithm for weighted partial maxsat. In *Proc. the 24th National Conference on Artificial Intelligence (AAAI'10)*. 2.4

[Argelich et al., 2008] Argelich, J., Li, C. M., Manyà, F., and Planes, J. (2008) The first and second MaxSAT evaluations. *Journal on Satisfiability*, 4:251-278. 2.4

[Berre, 2001] Berre, D. L., (2001) Sat4jmaxsat. In `www.sat4j.org` 2.4

[Eén and Sörensson, 2006] Eén, N. and Sörensson, N. (2006). Translating pseudo-boolean constraints into SAT. *JSAT*, 2(1-4):1-26 2.4

[Fu, 2007] Fu, Z. (2007). *Extending the Power of Boolean Satisfiability: Techniques and Applications*. PhD thesis, Princeton University. 2.4

[Fu and Malik, 2006] Fu, Z. and Malik, S. (2006). On solving the partial max-sat problem. In *Proc. of the 9th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT'06)*, pages 252-265. 2.4

[Heras et al., 2007] Heras, F., Larrosa, J., and Oliveras, A. (2007). MiniMax-SAT: A new weighted Max-SAT solver. In

Proc. of the 10th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT'07). pages 41-55. 2.4

[Li et al., 2009] Li, C. M., Manyà, F., Mohamedou, N. O., and Planes, J. (2009).
Exploiting cycle structures in Max-SAT. In *SAT'09*. 2.4

[Lin et al., 2008] Lin, H., Su, K., and Li, C. M. (2008).
Within-problem learning for efficient lower bound computation in Max-SAT solving. In *Proc. the 23th National Conference on Artificial Intelligence (AAAI'08)*, pages 351-356. 2.4

[Manquinho et al., 2009] Manquinho, V., Marques-Silva, J., and Planes, J. (2009).
Algorithms for weighted boolean optimization. In *SAT'09*, pages 495-508. 2.4

[Manquinho et al., 2010] Manquinho, V. M., Martins, R., and Lynce, I. (2010).
Improving unsatisfiability-based algorithms for boolean optimization. In *Proc. of the 13th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT'10)*, volume 6175 of *Lecture Notes in Computer Science*, pages 181-193. Springer.

2.4